Identification /5

Prénom: .....

Nom: .....

Code permanent: .....

Veuillez choisir **une seule réponse** par question, avec une croix (×) ou un crochet (✓).

Q1. Considérez une machine fonctionnant dans un environnement Linux, dont le répertoire courant contient le fichier `q1.c` suivant:

```
#include <stdio.h>

int main(void) {
  int c = getchar();
  return c == 'A' ? 0 : 1;
}
```

Qu'est-ce qui se produira si on entre la commande suivante?

```
$ gcc q1.c
```

- ☐ Le programme `q1` s'exécutera anormalement
- ☐ Le programme `q1` s'exécutera normalement
- ■ Un fichier `a.out` sera produit
- ☐ Un fichier `q1` sera produit

Q2. Considérez le fichier `q2.c` suivant:

```
#include <stdio.h>

int main(void) {
  putchar(getchar());
  return 0;
}
```

Supposons qu'on entre les commandes suivantes:

```
$ gcc q2.c -o q2
$ ./q2 < q2.c
```

Qu'est-ce qui sera affiché sur la sortie standard par la dernière commande?

- ☐ Rien
- ☐ `0`
- ■ `#`
- ☐ `q`

Q3. Qu'est-ce qui sera affiché sur la sortie standard par le programme suivant?

```
#include <stdio.h>

int main(void) {
  char c = -2;
  printf("%d\n", (unsigned char)c);
  return 0;
}
```

- ☐ `126`
- ☐ `-2`
- ■ `254`
- ☐ `%d`

Q4. Dans quel fichier devez-vous spécifier les valeurs `author` et `email` qui seront utilisées par Git lorsque vous allez créer des *commits*?

- ☐ `.git`
- ☐ `.gitignore`
- ■ `.gitconfig`
- ☐ `.gitlab-ci.yml`

Q5. Supposez que le répertoire courant contienne deux fichiers: le premier nommé `b` et le second nommé `Makefile`, dont le contenu est comme suit:

```
.PHONY: t

a: b
  cp b a

c: b
  cp b c

t: a c
  mv a d
```

Que se passera-t-il si on entre la commande suivante (indiquez l'énoncé qui est faux)?

```
$ make t
```

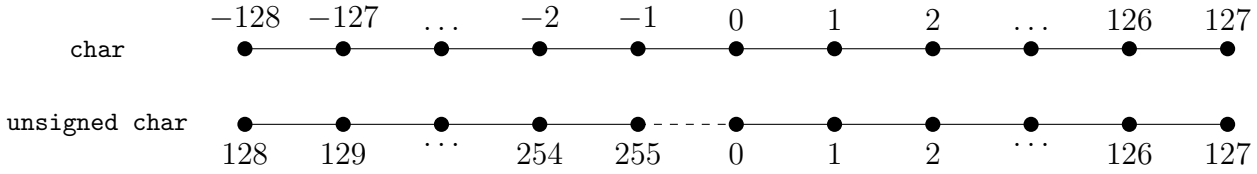
- ■ Le répertoire courant contiendra le fichier `a`
- ☐ Le répertoire courant contiendra le fichier `b`
- ☐ Le répertoire courant contiendra le fichier `c`
- ☐ Le répertoire courant contiendra le fichier `d`

Voir explications page suivante

Q1. La commande `gcc q1.c` sert à compiler le fichier source `q1.c` en un exécutable. Par défaut, en Linux, le nom de l'exécutable produit est `a.out`, alors qu'en Windows, c'est `a.exe`. Si on avait souhaité que l'exécutable soit nommé `q1`, alors il aurait plutôt fallu entrer la commande `gcc q1.c -o q1`. Il est important de ne pas confondre une commande qui compile un programme avec celle qui exécute un programme compilé.

Q2. Le programme décrit dans le fichier source `q2.c` lit un caractère sur l'entrée standard et l'écrit tel quel sur la sortie standard, puis termine son exécution. La commande `gcc q2.c -o q2` permet de compiler le programme en un exécutable nommé `q2`. La commande `./q2 < q2.c` lance l'exécutable `q2` en redirigeant le contenu du fichier `q2.c` sur l'entrée standard. Il va donc lire le premier caractère qui apparaît dans le fichier `q2.c`, qui est le caractère `#` de la ligne `#include <stdio.h>`.

Q3. Le type `char` permet de représenter une valeur entière entre -128 et 127, alors que le type `unsigned char` permet de représenter une valeur entière entre 0 et 255. Lorsqu'on convertit la valeur signée -2 en valeur non signée, on doit donc la ramener dans l'intervalle $[0, 255]$. On obtient donc l'avant-dernière valeur, soit 254. L'image ci-bas illustre visuellement la correspondance lors de conversions entre les types `char` et `unsigned char`.



Q4. Le fichier `.gitconfig`, qu'on place dans le répertoire de connexion `~`, sert à configurer l'utilisation globale de Git par un utilisateur donné. On y renseigne notamment son identité (`author` et `email`), mais on peut aussi spécifier d'autres éléments, comme des synonymes (*alias*) et des comportements à adopter lors d'une connexion réseau. Le fichier `.git` est un répertoire, qui contient tout l'historique. Il est donc spécifique à chaque projet. Le fichier `.gitignore` permet de spécifier des motifs (*glob*) des fichiers qu'on souhaite ignorer lorsqu'on utilise Git, c'est-à-dire qu'on ne veut pas en faire le suivi dans l'historique. On peut en mettre un par répertoire d'un projet. Finalement, le fichier `.gitlab-ci.yml` est utilisé pour définir des pipelines qui sont exécutés dans des conteneurs Docker de façon automatique lorsqu'on pousse des modifications sur la plateforme GitLab. Il peut notamment être utilisé pour vérifier que la base de code compile correctement et que les tests fonctionnels réussissent. Il doit être placé à la racine du projet pour que GitLab puisse le retrouver.

Q5. La commande `make t` demande à l'outil Make de produire la cible `t` du fichier `Makefile`, c'est-à-dire la troisième cible. Pour la produire, il faut d'abord produire les cibles `a` et `c`, qui sont des dépendances de cette cible. Les cibles `a` et `c` dépendent toutes deux de la cible `b`, qui est un fichier qui existe dans le répertoire courant. Elles sont produites à l'aide des recettes `cp b a` et `cp b c`, qui copient toutes deux le fichier `b` vers des fichiers `a` et `c`. Une fois les cibles `a` et `c` réalisées, on peut produire la cible `t` en exécutant la recette `mv a d`, qui renomme le fichier `a` en un fichier `d`. On obtient donc, à la toute fin, un répertoire contenant les fichiers `b` (qui y était déjà), `c` (obtenu en copiant `b`) et `d` (obtenu en renommant `a` par `d`). C'est donc le premier énoncé qui est faux, puisque `a` n'existe plus, ayant été renommé.